A Model of Successful Patterns

of Progress During the Integration of Software

by

MARY LOU A. LANCHBURY

B.S., Kansas State University, 1985

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

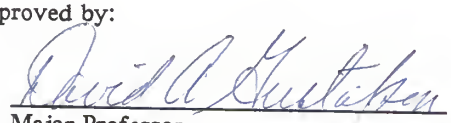
Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1986

Approved by:



Major Professor

ACKNOWLEDGEMENTS

I would like to thank Robin Niederee for her help in getting this thesis into a readable form — especially the tables.

Thanks also to my children, Ben and Naomi, for excusing their mother's absences from family life while she was trying to finish, and to Tim for his patience and suggestions.

And thanks to Dr. David Gustafson for all the help, suggestions and direction he has given throughout the course of this research and thesis.

LD
2668
.T4
1986
L36
c. 2

A11202 971169

Table of Contents

Acknowledgements i
Chapter One - Introduction 1
Chapter Two - Background 4
2.1 Presentation of Software Development Cycle and Software Life Cycle 4
2.1.1 Requirements Analysis 4
2.1.2 Program Specification 5
2.1.3 Design 6
2.1.4 Coding 6
2.1.5 Unit Testing 6
2.1.6 Integration of Unit Modules 7
2.1.6.1 Top-Down Integration 7
2.1.6.2 Bottom-Up Integration 8
2.1.6.3 Sandwich Integration 9
2.1.7 Validation 9
2.2 Literature Survey 10
2.2.1 Cost Estimation 10
2.2.2 Staffing Levels 12
2.2.3 Scheduling 13
2.3 Background of Data Collected and Used 14
Chapter Three - Analysis Methodology 16
3.1 Changes by Statement Type 16
3.2 Hierarchical Changes 17
3.3 Complexity Measures 19
3.4 Summary 21
Chapter Four - Model for Ideal Case 22
Chapter Five - Case Studies 24
5.1 Successful Case 24
5.2 Unsuccessful Case 26
Chapter Six - Successful Case Study vs Proposed Model 28
6.1 Amended Model 28
Chapter Seven - Conclusions 30
Bibliography 32
Appendix A - Successful Case Data 35
Appendix B - Unsuccessful Case Data 39
Appendix C - Manager's Model for Progress Evaluation 43

Chapter One

Introduction

Data on patterns of progress during the software development cycle is a rare thing. While there is some macro data available, detailed data about progress patterns has not been gathered or analyzed. Without historical and detailed data to use as a guide, estimation of time schedules, delivery dates and software costs has been "seat of the pants", with poor results. These estimates of time and money are set but rarely kept. "Seeing" progress during software development is very difficult. Procedures, techniques, strategies and tools that could provide visibility of progress are not generally available - particularly for the non-technical project manager. The ability to evaluate the progress of a project during the different phases of development is necessary in order to determine if adjustments to the project's costs and time schedules are necessary.

Middle- and upper-level managers with no background in software or its development, are often given responsibility for software projects. Technical personnel involved with the actual project production and who have background experience with software and its development, can find it difficult to communicate with the non-technical personnel about the progress of a project (or for that matter other aspects of the project). Milestones regarding progress that are comprehensible to the technical members tend to be incomprehensible to the non-technical members of the project team. Milestones also occur far enough apart that visibility of progress is hindered. These non-technical members are often put in charge of project decision making and they do not have the tools/data necessary to make these decisions analytically. Therefore, any representation of progress needs to be understandable and usable by both technical and non-technical personnel alike.

This thesis presents a model of successful code change patterns during the integration phase of the software development cycle (the integration phase is defined and discussed in Chapter 2). Models generally may be based on either an empirical or a theoretical approach. The empirical approach uses data gathered from previous projects to evaluate the current project. The theoretical approach is based on assumptions about such things as how people solve problems. The model presented here is empirically derived from a successful project; it is contrasted with failed project data to highlight those characteristics that empirically and intuitively should distinguish a successful development pattern from possibly unsuccessful development patterns. The model does not claim to be the ONLY model of successful patterns during integration. It is a basis from which to expand both in terms of a larger database to support it and the addition of other facets for analysis. It also presents a method of progress evaluation that can be used by ALL personnel. Other successful patterns very well may exist and future research may uncover them.

The model should enable project personnel, the non-technical personnel in particular, to visualize the progress of the integration phase of the project in an objective manner. A factual basis that can be used for decision making about the extension of deadlines and/or the expansion of budgets is needed. The model should be understandable, clear, usable, objective and provide visibility of progress in the form of patterns developing from the application of change measures to the data. These characteristics should be applicable for non-technical and technical personnel alike. While using technical concepts and tools, the results of the model presented in Chapter 4 are non-technical enough in the presentation of progress evaluation to be understood and used by personnel who may or may not have a technical background. Appendix C presents a check-list format of the model to aid managers in the use of the model.

Chapter 2 introduces the concept of software engineering, the software develop-

ment cycle, the software life cycle and other background information. Chapter 3 discusses the analysis tools chosen for use. After the model is discussed in Chapter 4, case studies are presented in Chapter 5 and then contrasted with the model in Chapter 6. The model is then revised to a working model in Chapter 6.

It is important that the managers of software projects start to move away from its current "seat of the pants" methodology into a visible progress methodology. Hopefully, while the model of progress presented is for only a part of the software development cycle, it will open the door for development of models of progress for the other phases.

Chapter Two

Background

2.1 Presentation of Software Development Cycle and Software Life Cycle

During the past decade software costs have risen dramatically, becoming the single largest expense in many computer based systems. Software is the system element that is most difficult to plan, least likely to succeed on time and within cost, and the most dangerous to manage. Software engineering techniques involve planning, analysis, design, implementation, testing and maintenance, and form the basis of a software engineering methodology that is application-dependent. The main objectives are 1) a well-defined methodology that addresses a software life cycle of planning, designing, development and maintenance, 2) a defined set of software components that provides documentation of each step in the life cycle and shows traceability from step to step and 3) a set of milestones [Pressman1982].

Two cycles are associated with software production. One is the software development cycle and the other is the software life cycle. The software development cycle is a subset of the software life cycle and will be discussed first. Although there are slight variations on the nomenclature of the phases, the phases in the software development cycle are as follows:

1. Requirements Analysis
2. Program Specification
3. Design - Preliminary and Detailed
4. Coding
5. Unit Testing
6. Integration of Unit Modules
7. Validation

A definition and general information for each stage follows.

2.1.1 Requirements Analysis

The requirements analysis phase is the starting point of software development. This phase is also known as the problem definition phase. During the requirements analysis phase the user's requirments are carefully examined so that the intent of the desired system, the properties it should possess, and any constraints on it are identified. The user's needs versus the user's desires are differentiated.

The purpose of this phase is to identify the problem and specify a solution that will fulfill the user's needs. The solution also needs to be precise enough to be produced by the software project team.

Typically, products of this phase include a Software Plan, a preliminary user's guide, optional prototype development, and quality assurance and verification plans. These products should be complete enough to answer all questions regarding what work is to be done and if necessary, serve as a contractual agreement between the developer and the customer.

2.1.2 Program Specification

The program specification phase is an extension of the requirements analysis phase. A precise specification for the desired software system is formulated. This specification is prepared in terms that are understandable and can serve as guides for designers, coders and testers. It is an interpretation of the requirements documents. The Software Requirements Specification that is produced should be cross-checked with the products of the requirements analysis phase to ensure accuracy and correctness. Cross-checking this document with the user is also valuable for verification purposes.

2.1.3 Design

Design involves analysis of the Software Requirements Specification. The flow of the data and important algorithms are planned and the system is organized into modules. The design phase can be broken into two sub-phases: preliminary design

and detailed design. During preliminary design, the modules are identified and the software structure or hierarchy is defined. A precise specification of what each of the modules in the software hierarchy does is done in the detailed design sub-phase. These specifications are also called module specifications. Some of the areas that are explicitly specified include module interfaces, externally observable behavior, input/output parameters, global data accessed, and the project team member responsible for the module. The more formal the specification is, the more likely correct implementation is to occur.

2.1.4 Coding

During the coding phase of the software development cycle the design is translated into a machine understandable form. Each module is coded following the algorithm produced during the design phase and with close attention to the module specification(s) for that particular module.

Although this phase is the one most associated with system development by the lay person, it actually only constitutes about 15% of the total development time expended [Gilbert1983]. It should be a direct follow through of the design phase.

2.1.5 Unit Testing

A critical element of software quality assurance is software testing. Software testing is the ultimate review of specification, design and coding. It is, at least psychologically, a destructive process because successful testing discovers errors. Error free-ness cannot be proved by testing. In this discussion of the software development cycle, testing is broken into three sections: unit testing, integration of unit modules and validation.

Individual modules are tested to see that they function properly as an individual unit. Control paths should be tested to uncover errors within the boundary of

the module. The unit test is always white-box oriented. In white-box testing the internal workings of the module are known and can be tested to assure that they perform according to the specifications.

2.1.6 Integration of Unit Modules

This is the phase of the software development cycle which is examined in the development of the model of progress that is presented in Chapter 3 and for which the model is designed to analyze.

During this phase unit modules are integrated together into an overall system. Interfaces are checked for correctness in terms of parameters, affects of global variables are checked and correctness of variable initialization is checked. These are just a few of the areas of testing that are done during integration. Integration is a systematic technique for assembling software while testing to uncover errors associated with interfaces.

The most important and most time-consuming system errors are interface errors, i.e., interaction errors between modules. Experience has shown that software integration requires a large amount of time because of errors that arise in the transfer of information between modules.

In the literature there are three approaches to integration. They are:

1. Top-Down Integration
2. Bottom-Up Integration
3. Sandwich Integration

2.1.6.1 Top-Down Integration

To overcome the time problem of integration, most developers use the strategy of top-down integration and testing. Top-down integration is an incremental approach to system assembly. The top-down integration process is performed in a series of steps:

1. The main module is used as a test driver with modules immediately subordinate to it being "stubbed" in.

2. Stubbed in modules are replaced one at a time with actual modules.

3. As each module is integrated, testing occurs.

4. Regression testing may be done to make sure new errors have not been introduced with the replacement of a stub with an actual module.

Top-down integration offers several advantages. Theoretically, errors are localized to the new modules and interfaces that are being added to replace stubs. The top level modules provide a test environment for the lower level modules. Since major control or decision points are encountered early in the system, they are also verified early in the testing.

Top-down integration sounds fairly uncomplicated but in practice causes loss of correspondence between specific tests and incorporation of specific modules. No significant data can flow upwards from the stubs so many tests must be delayed until replacement of stubs with actual modules occurs. While it seems that errors would be localized to new modules and interfaces being added, the fact remains that since many tests must be delayed until this addition, the upper modules may also still contain errors.

2.1.6.2 Bottom-Up Integration

Bottom-up integration assembles and tests modules starting with modules at the lowest levels in the software structure or hierarchy. Stubbing is eliminated since subordinate modules are always available. As individual modules pass their unit testing, they are combined into sub-groups. The sub-groups are tested and then combined into larger sub-groups until finally the entire system is put together from these sub-group pieces. Normally each of these sub-group pieces represents a major section

of the total system. Control and data interfaces require extensive testing with test cases being carefully chosen. Exhaustive testing of sub-systems is not practical or feasible because of the increasing complexity of the sub-systems and their interfaces. This complexity problem is readily visualized in the case of the "big bang" approach to bottom up integration in which after unit testing, ALL modules are linked and executed in one single integration step. Isolation of error sources is a true headache in this case.

2.1.6.3 Sandwich Integration

The sandwich integration strategy [Fairley1985] is a combination of the top-down integration strategy and the bottom-up integration strategy. It is predominately top-down but bottom-up strategy techniques are also used. Individual modules and sub-systems are built using the bottom-up integration strategy. Integration of the sub-systems into the system in its entirety is done by using the top-down integration strategy. Thus indiviudal modules and sub-systems are tested prior to replacement of stubs. The advantages of the top-down integration strategy are retained while some of the problems are eased.

2.1.7 Validation

Validation is said to succeed when the system performs in the manner that it is reasonably expected to. Reasonable expectations should have been defined in the Software Requirements Specification in a section entitled Validation Criteria.

Validation testing occurs after the software is completely assembled as a package. It consists of a series of black box tests demonstrating requirements conformity. A test plan and a test procedure are designed. These check that all functional requirements and performance requirements are met, the documentation is correct and geared to human understandability, and any other types of requirements are satisfied (e.g., maintainability, etc.). The test plan develops the types of tests to be

done. The test procedure develops the specific test cases. After each test case is run, one of two states occur. Either the specifications were met and the function or performance characteristic is accepted or there was a deviation from the specification. When a deviation is detected it is noted. Most deviations are not correctable prior to delivery and must be resolved thereafter.

As indicated earlier, two cycles are associated with software production. The software life cycle is identical to the software development cycle with the addition of one more phase — the maintenance phase. Maintenance involves the updating of code and documentation to meet changes in the user's requirements. Resolution of unresolved deficiences and "bug fixing" are likely to be part of maintenance also. When updating or fixing becomes too extensive, the software system should be replaced or the existing system reconfigured.

Previous analysis of the change patterns occurring during the maintenance phase provided the basis for the research done for the integration phase [Gustafson1985]. This work analyzed the types of statements being changed during the maintenance phase. This analysis technique is further discussed in Chapter 2.

2.2 Literature Survey

Generally speaking, the literature tends to emphasize mathematical estimation models and formulas when looking at management of software development. Cost estimation, staffing levels and scheduling are some of the areas that are heavily explored in the literature.

2.2.1 Cost Estimation

Estimating the cost of a software project is one of the hardest tasks in software development. The major factors influenciing software cost are:

1) Programmer ability
2) Product complexity
3) Product size
4) Available time
5) Required reliability
6) Level of technology [Fairley85]

Harold Sackman [Sackman68] conducted an experiment in 1968 relating to programmer ability. The goal of the experiment was to find out the influence on programmer productivity of batch and time-shared access. The results showed a much wider difference in programmer ability than could be attributed to the machine access method. He showed that programmer ability is a significant factor in cost estimation from this experiment.

Frederick Brooks in his book "The Mythical Man-Month" states that utility programs are much more difficult to write than application programs and systems programs more difficult than utility programs [Brooks74]. Boehm extends Brooks' evaluation and produces mathematical equations to ultimately predict programmer cost for each type of program [Boehm81]. These equations are empirically derived from a large database. Boehm's equations are aimed at small to medium sized projects where the environment is familiar and well understood. Large projects obviously are more expensive to develop than small ones.

While Boehm's figures can be adapted for total project effort, Putnam directly addresses the issue of total project effort [Putnam78]. Putnam's model uses linear programming techniques for projection of development schedules. Putnam feels that there is a point of nominal schedule compression that cannot be surpassed regardless of the number of people or resources utilized. Putnam's estimation model is based on the Rayleigh-Norden [Norden77] curve which shows the distribution of effort over the software life cycle. The Rayleigh-Norden curve can also be used to mathematically relate the number of delivered lines of code to effort and development time [Norden77].

Software reliability is the probability that a program will perform a required function under stated conditions for a stated period of time [Fairley85]. Reliability is often expressed in terms of accuracy, robustness, completeness and consistency of source code. Boehm addresses the issue of reliability by designating categories of reliability and establishing development effort multipliers for each category [Boehm81].

Cheung states that program reliability is based in how reliable the modules are and examines the effect module change has on the reliability of the whole program [Cheung80]. Musa bases his model of software reliability on execution time [Musa80]. Littlewood and Verrall present a Bayesian reliability growth model based on the frequency of failures of a piece of software [Littlewood73]. There is little consensus on exactly how to measure reliability.

Boehm handles the level of technology issue by again providing effort multipliers based on the level of technology available [Boehm81].

Number of lines of code has also been used to base cost estimates on. However, this method is not very reliable and needs to be cross checked with some other method.

By far, however, the most widely used cost estimation technique is expert judgment. Expert judgment relies heavily on the background experiences of the person(s) doing the cost estimation [Fairley85].

2.2.2 Staffing Levels

As a project progresses the level of staffing changes. The Rayleigh-Norden curve is representative of the staffing level at specific points in development [Norden77]. Putnam's model also addresses estimates of staffing as does Boehm's model [Putnam78, Boehm81].

Esterling's productivity model is based on the microscopic characteristics of the work environment such as number of interruptions in a day, average recovery time

after an interruption, etc. [Esterling80]. He develops an empirical relationship for the fraction of useful working time per work day per person.

Walston and Felix isolated 29 factors that they felt affected productivity [Walston77]. They then attempted to show how productivity varies with each factor by deriving a set of single variable models for effort, project duration, pages of documentation and staffing levels. These models are functions of the number of lines of code and are environment- and application-specific. They cannot be applied generically.

One of the most common fallacies in estimation of staffing levels is described by Brooks in "The Mythical Man-Month" [Brooks75]. Managers often believe that if they fall behind schedule they can catch up by adding more people to the project. Brooks says that adding people to a late project will make it later.

Expert judgment also appears to be one of the most widely used methods for estimating staffing levels.

2.2.3 Scheduling

The estimate of the amount of time it will take to complete a project can be difficult and must often be revised. A number of techniques and models exist to help estimate a project's completion time.

Basili and Zelkowitz empirically derived equations to predict person-months and total weeks of effort needed for a project [Basili79]. Their equations are based on medium to large scale systems.

The program evaluation and review technique (PERT) and the critical path method (CPM) are scheduling methods that develop a network defining pictorially the tasks to be accomplished and in what order they need to be done [Pressman82]. Boundary times for a given task are established to help evaluate progress.

Little of the literature specifically addresses the issue of general management of the software development cycle. Most models and/or mathematical equations presented to aid in project management only handle a very specific issue (such as cost estimation) and in order to be used, require a technically oriented manager. There is a lot to be said for placing technically oriented people in project management positios. Past experience shows this is not happening. Thayer, Pyster and Wood [Thayer81] in investigating the major problem areas in software project management found that procedures and techniques for project manager selection are poor. They also found that success criteria is frequently inappropriate and procedures, techniques, strategies and aids that provide visibility of progress to the project manager are not available.

Almost all models pertaining to management of software development are very mathematically and technically oriented. We often tend to forget that software development is still partially an art and the "art side" needs to be modelled also. Non-technical management personnel need to be provided with usable and understandable models for management of software development. The literature shows this has not been done.

2.3 Background of Data Collected and Used

The data used for this research consisted of modules and programs written in the language "C" and implemented on the 8/32 under UNIX. On a daily basis during the final days of the integration phase "snapshots" of the modules and programs were taken. For each module and program there exists a collection of versions of the code that depicts the integration activities.

The modules and programs were developed and written by junior and senior computer science/information science students enrolled in CMPSC 341 Software Engineering Project II. This course is a required course in the undergraduate curriculum in the Computer Science Department at Kansas State University, Manhattan, Kansas. It is a continuation of CMPSC 340 Software Engineering Project I. The

techniques of software engineering are taught and the application of these techniques is accomplished through the development of the projects assigned to the students. There were seven teams for which data is available. From these seven, the two with the most complete data sets were chosen to evaluate completely and to include in this research.

Chapter Three

Analysis Methodology

Once the data was assembled, objective change measures that would show patterns of progress were needed. The patterns of progress needed to be definite and clear. Non-technical personnel as well as technical personnel need to be able to use and understand the patterns. After examining the different aspects of the data such as completeness, style, etc., various change measures were evaluated. The change measures chosen needed to be reflective of the goals of the model including understandability, usability, clearness, exhibition of patterns, and objectivity. Three change measures were chosen to use in the analysis and definition of the model of progress for the integration phase. The three change measures chosen were:

1) Changes by statement type

2) Changes within the software hierarchy or structure

3) Changes in complexity

3.1 Changes By Statement Type

The initial measurement approach selected was to analyze the types of statements being changed. This was a logical starting point since a similar analysis of the types of statements changing during the maintenance phase suggested this research [Gustafson1985].

The types of statements being changed would be reflective of the work done during the earlier development phases. Statement changes reflective of the goals of the integration phase (interfaces, etc.) would be indicative of the addressing of integration phase issues and hopefully their resolution. Statement changes that reflected poor design or incomplete design specifications would indicate a lack of

integration progress since in reality, development would still be continuing.

To accomplish the changes by statement type analysis, a tool was used that had been developed for analyzing changes during the maintenance phase. This tool compared two source code files for differences using the UNIX utility program "diff". After the comparison, the lines of the first file were prefixed with code letters indicating change in that line (if a change had occurred).

When a module or program had changed from one version to the next, this tool was run on the two source code files to identify which statements had been changed. The file with the prefixed lines was then sorted by statement type. A total count for occurrences of each statement type was found and the total number of each statement type that was changed was found (i.e., 100 "if" statments of which 8 were changed). The percentage of change for each statement type was then figured (total changes of a statement type divided by total number of that same statement type).

These figures showed which types of statements were changed most frequently during the integration phase. For a given statement type it could be stated that X% of this type of statement changed during the integration phase of the project.

3.2 Hierarchical Changes

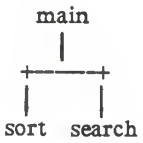
The location of changes within the software hierarchy or structure was also examined for the data sets. If a pattern in the location of the changes within the hierarchy could be found, it would be indicative of the technique of integration actually being used (the actual technique may be different from the intended integration technique). If no pattern could be found this would reflect a serious disorganization of the integration phase and a need for corrective action.

To evaluate the location of changes within the software hierarchy, the actual software hierarchy itself had to be identified. Although hierarchy diagrams for each project existed, it was felt that a more valid hierarchy would be obtained from the

source code itself — a case of what is said to be done versus what is actually done. To do this a shell program was written using the UNIX utilities "sed" and "awk" to remove all lines of code that did not contain a module call. The lines of code left would enable one to build a hierarchy diagram since the module would have different levels of indentation. The module declarations or headers were flush with the left margin while any calls made by the module were indented from the left margin. Consider the following example of a stripped file:

```
main
   sort
   search
sort
search
```

The module "main" calls the modules "sort" and "search". Neither of the modules, "sort" or "search", make any calls to other modules. Therefore the software hierarchy could be constructed as follows:



This part of the analysis was done by hand although one of the projects currently being developed by the Spring 1986 CMPSC 341 Software Engineering Project II class is a tree drawer that would generate a hierarchy structure from the type of file described earlier. Hierarchy diagrams were built for each change period.

The files produced in the changes by statement type analysis were then re-analyzed. This time a percentage change for each module was found. Percentages for modules showing the most significant change were then associated with their respective modules in the hierarchy diagrams. As the changes from one change period or delta to the next were recorded, the successive hierarchy diagrams were examined for

integration patterns.

A top-down integration pattern would be associated with top level modules having the higher percentages of change in the initial change period and these higher percentage positions appearing at subordinate levels as integration progressed. Also expected would be a downward growth of the hierarchy as more modules replaced the stubs.

A bottom-up pattern would show the lowest level modules exhibiting the higher percentages of change in the initial change period with an upward progression of the higher percentages of change. The hierarchy would also be expected to grow both horizontally and upwards as sub-systems were added and additional pieces were integrated.

A sandwich integration would show an ordered pattern that at first may appear to have no pattern of integration. Lower modules would exhibit higher percentages of change in the early change periods. Horizontal and upward growth would also appear in early diagrams. At some point the entire structure would be available and the change percentages would go through a top to bottom progression as in the top-down integration pattern.

Some integration technique should be identifiable for each project from the progression through the hierarchy in terms of growth and/or percentage of module change.

3.3 Complexity Measures

A measurement of the complexity of the programs and modules was also used to look for patterns. Complexity measures are designed to measure the complexity or understandability of a design/program. Some controversy arises as to what exactly a particular complexity measure means in terms of what it measures. However, for this research, complexity measures were chosen for data analysis in order to deter-

mine patterns of code change and not to provide an understanding of the program's complexity.

McCabe's cyclomatic complexity measure [McCabe1976] and Halstead's software science measure [Halstead1977] were chosen as the two complexity measures to use in the data analysis for several reasons. First, both have been around long enough for people to be familiar with them and for the complexity measures to have some kind of visible longevity. Second, and probably most significantly, automated tools were available to perform the two complexity measures. This eliminated the necessity of performing these calculations by hand and thus introducing human error. Finally the two complexity measures have been shown to differ from each other based on the type of project they are analyzing. [Mata84, Mata86] Thus if one does not show a pattern, the other might. However, if neither of the complexity measures show a pattern it would indicate that a pattern very likely did not exist.

Part of the analysis of the complexity measures dealt with inter-module versus intra-module complexity or the complexity of the whole versus the complexity of the parts. The inter-module complexity or the complexity of the integrated system would be evaluated for patterns in accordance with the integration technique (top-down, bottom-up or sandwich) identified in the hierarchical change analysis. Both top-down and bottom-up would show a significant increase in overall complexity as integration progressed. The increase in complexity during a sandwich integration would not be as significant as the increase during top-down or bottom-up integration and would level out prior to the end of the integration phase when all modules were in place.

Intra-module complexity would not be expected to fluctuate significantly throughout the integration phase. Individual modules have been unit tested prior to the integration phase beginning and should be stable in complexity.

The automated metric tools were run on the programs for each delta or change period. The results for each complexity measure were examined to note any increase or decrease in the complexity of the whole program and of the individual modules. Patterns of increase or decrease in complexity were identified.

An added feature that arises from the different change measure evaluations of individual modules is the highlighting of modules that may be having significantly more integration problems than others. This would help isolate problem modules. The reasons behind the individual module's integration problems could then be evaluated. Since this evaluation would require action by technical personnel in order to correctly analyze the reasons for the problem, highlighting of individual modules with problems is mentioned here; it is not incorporated into the model of successful patterns since it would be difficult for non-technical personnel to utilize successfully.

3.4 Summary

The three change measures were all evaluated for patterns (or lack of patterns) during the integration phase. These patterns were compared with a model for the ideal case which had been developed based on experience and intuition. From the patterns identified, a revised model for successful patterns during the integration phase of software development was built. Unsuccessful trends suggested by the change measures of the data will be included to aid in delineating lines of progress since other successful patterns of progress may be found as further research is done.

Chapter Four

Model for Ideal Case

Before evaluating the data, it was deemed appropriate to fashion a model for the case of ideal progress during integration. This would give a intuitive foundation to base our observations on and also a hypothetical model to evaluate results against.

Few changes should be needed to the code during the integration phase of the software development cycle if the project has been defined and designed precisely enough including complete module specifications.

The three change measures that had previously been decided upon as the basis for the model of the integration phase were each evaluated for patterns for the case of ideal integration.

Regarding types of statements being changed, it was felt that while few changes should be occurring, those that did would probably be attributable to the actual integration itself. These types of statements would include subroutine calls and system calls. Declaration statements should not be changing since data structures should have been set much earlier in the design phase of the project. Some looping structures may show change but this should be traceable to a change in one of the integration related statements, or to possibly an enhancement related to speed of execution.

The location of the changes within the hierarchy structure should follow a top-down approach. This is the method of integration taught in the computer science curriculum and in particular in the lecture portion of the software engineering project class.

The complexity of the code for the overall system would be expected to increase dramatically as individual modules are integrated. However, the complexity of the

intra-module code would not be expected to increase or decrease significantly — probably only very slightly. A small amount of change would be expected. This would indicate that the same statements were not being repeatedly changed with no progress occurring.

The model for the ideal case of progress during integration of software discussed above, presents the change patterns that should occur during successful integration. These change measure patterns will be compared with the actual data after its presentation.

Chapter Five

Case Studies

Although a large amount of data was available and was analyzed during this research, there were two teams whose data was chosen for presentation in the case studies section of this paper. The first team, G3, represents the successful case. They were able to finish their software development project on time while meeting their specifications. Their data exhibited identifiable patterns of progress during the integration phase. The second team, G5, represents the unsuccessful case. This team did not finish their project and did not seem to be making progress towards completion. Their data exhibited a randomness which was not identifiable with anything — especially progress. Thus both ends of the spectrum are represented by the data presented here.

5.1 Successful Case

Team G3 completed their test coverage project on time meeting their defined requirements. They had a total of nine individual modules. Four versions were examined representing the three change periods that occurred during G3's integration period.

The statement type exhibiting the highest percentage of change during the integration phase for G3 was subroutine/system calls. Twenty-one (21) percent of all subroutine/system calls were changed during integration. Assignment statements also exhibited a fairly significant amount of change with eighteen (18) percent of all assignment statements changing during integration. However the percentage of change for assignment statements steadily decreased during the integration phase. Thirteen (13) percent of if statements were also changed. All other types of statements showed less than thirteen (13) percent change. The pattern of statement

changes tended to show an increase in the second change period followed by a return to the same general percentage range as the first change period. Percentages for overall changes in all statement types are presented in Appendix A, Table 1. The raw data for types of statements changed is presented in Appendix A, Table 2.

Hierarchical analysis revealed some rather surprising results in that a strict application of any of the three integration strategies was not used. Instead all modules were put together into one system similar to the starting point of the bottom-up "big bang" technique. However, from there on the bottom-up integration technique was utilized. At first, the lowest level modules exhibited the highest percentages of change indicating the occurrence of their integration. This was followed by an upward pattern through the hierarchy of the higher percentages of change. Appendix A, Table 3 presents the percentages of change for each delta or change period. The hierarchy itself was stable during the integration phase.

McCabe's cyclomatic complexity measure (see appendix A, Table 4) remained relatively stable for both inter-module complexity measures and intra-module complexity measures throughout the integration phase. The intra-module pattern followed the expected pattern of stability that was put forth in the proposed model. The stability of the inter-module pattern is not unreasonable given the integration technique used.

Halstead's software science measure (see appendix A, Table 4) showed more change than the McCabe's complexity measure. At the intra-module level, change ranged from no change at all to a large change - in some cases doubling from one change period to the next. The inter-module figures showed an increase overall of about forty (40) percent.

The successful pattern of integration exhibited by G3 showed subroutine/system calls and assignment statements changing most frequently. A "big

bang" start in the hierarchy analysis followed by a bottom-up integration strategy was observed. McCabe's cyclomatic complexity measure showed stability at the intra-module level and the inter-module level. Halstead's software science measure did not show any pattern of stability.

5.2 Unsuccessful Case

Team G5 did not complete their test coverage project and did not show any signs of progress towards integration. They had a total of eleven individual modules in their project. As with team G3, four versions were examined which represented three change periods that occurred before team G5's project was abandoned.

Almost every type of statement was changed significantly during G5's integration phase (see appendix B, Tables 1 and 2). Forty-five (45) percent of the statement types exhibited a change of thirty (30) percent or more. Assignment statements, subroutine/system calls, declarations, breaks, and case statements all exhibited percentage changes of greater than thirty (30) percent. If statements and while statements showed change percentages between twenty-two (22) percent and twenty-seven (27) percent. All other statement types had less than a fourteen (14) percent change. The fact that almost thirty-four (34) percent of all declaration statements were changed indicates some significant design deficiencies. Appendix B, Table 1 presents the percentage change for each statement type for G5's integration phase. Appendix B, Table 2 presents the raw data for the statement type changes.

Team G5's statements exhibited a much higher percentage of change overall than did team G3. The types of statements being changed and their high percentage of change indicates significant design deficiencies and lack of progress towards project completion. Team G5 moved from one software development phase to the next prior to completing that phase. The premature movement into the integration phase was probably only one in a series of premature phase advancements.

Changes in the hierarchy also exhibited no pattern. Although the hierarchy itself was stable, the integration technique, if one was used, escaped identification. Appendix B, Table 3 presents the module change data.

McCabe's cyclomatic complexity measure (see appendix B, Table 4) shows virtually no change at all at either the inter-module or intra-module levels. While this might seem to point to progress, it probably does not. Instead, it indicates that the same statements are probably being changed over and over with no progress being made towards integration.

Halstead's software science measure (see appendix B, Table 4) appears relatively stable at the inter-module level but this can be explained by examining the intra-module level complexity figures. Fluctuations from one change period to the next can be seen with both increases and decreases occurring. These increases and decreases appear to offset each other and give the illusion of inter-module stability.

G5 exhibited a mishmash of inconsistencies and wild variations in change. No patterns were identifiable probably due to the lack of an integration plan or procedure. While team G3 was successful and progressed through the integration phase, team G5 not only wasn't successful or progressing, but must have been working with an extremely incomplete software development base of prior phases. It was not clear that their hierarchy structure was workable, their data structures had not been clearly defined and they appeared not to have any direction towards progress. Team G5 wandered around lost in their integration phase. It is apparent that G5 should have returned to their previous phases - particularly their design phase and clarified essential design issues before progressing on.

Chapter Six

Successful Case Study vs Proposed Model

The successful case study presented supported the proposed model's assertion that subroutine/system calls would exhibit the most significant change in statement types. However the fact that assignment statements also changed significantly was not foreseen in the proposed model. Further work investigating the reason(s) for the significant change in assignment statements might prove beneficial.

The location of the changes within the hierarchy structure of the successful case study did not support the proposed model's top-down integration strategy. It indicated a "big bang" start followed by a bottom-up integration strategy. This strategy will be referred to hereafter as "G3's big bang integration strategy".

Since the proposed model's location of changes in the hierarchy pattern was not supported, neither was the inter-module complexity pattern. The inter-module complexity pattern for the proposed model showed complexity increasing as integration progressed. The successful case study exhibited a smaller increase in overall complexity than was expected from the proposed model. Relative stability was also observed in the successful case study.

Intra-module complexity patterns for the proposed model were supported by the successful case. Individual modules exhibited stable complexity patterns in both the successful case study and the proposed model.

6.1 Amended Model

After comparison of the proposed model and the successful case study, an amended model for successful patterns of integration was developed.

Subroutine/system calls are still the statement type expected to exhibit the most significant change. Further analysis may support the inclusion of significant

changes in assignment statements. At this point changes in assignment statments will be included as a note to but not part of the model. Other statement types should exhibit little or no change during the integration phase.

Declaration statements including define statements should be representative of data structures that were thought out and defined in earlier phases. These statement types should show very little or no change during the integration phase. Likewise decision structure statements such as case statements and if statements, and returns, breaks, and else statements should show little or no change during integration. These statement types represent the algorithmic structure of the project which should have been defined in an earlier phase.

The change percentages for comment statements should be relatively low. The comments should be reflective of the overall algorithm and debugging should be aligning the code with the comments.

A stable hierarchy should evolve during the integration phase. An identifiable integration technique should be found. However, which integration technique used will be dependent upon the situation and the project team members. Top-down integration strategy, bottom-up integration strategy, sandwich integration strategy or the G3 big bang integration strategy might be identifiable.

The pattern of the complexity measures for the inter-module measures will be dependent upon the integration strategy identified in the location of change in the hierarchy analysis. The top-down integration strategy and the bottom-up integration strategy would exhibit dramatic increases in inter-module complexity. Sandwich integration strategy would indicate a visible increase early with a leveling off as integration progresses. The G3 big bang integration strategy would exhibit inter-module stability throughout the integration phase.

Intra-module complexity should remain relatively stable throughout the integration phase regardless of the integration strategy identified.

Chapter 7

Conclusions

The proposed model for successful patterns during integration utilizes patterns for three change measures to show integration progress or lack of progress for the programs/modules of the project. First, the types of statements being changed are analyzed. Subroutine/system calls should show the highest percentage of change with the changing of assignment statements not indicating an unreasonable behavior. The changing of a high percentage of declarations would indicate a need for the return to earlier phases of development to complete or re-do work there. High percentages of change in other types of statements would not be indicative of progress in the integration phase according to this model.

Second, the software hierarchy or structure should become stable. The progression of change in the modules in the hierarchy should have an identifiable pattern that can be associated with one of the integration strategies discussed in previous chapters. An inability to identify a pattern of integration or an unstable software hierarchy would indicate lack of progress in integration.

Finally, depending on the integration strategy identified in the second part of the model, a pattern in the increase/decrease of the complexity measures should be apparent. Using McCabe's cyclomatic complexity measure and Halstead's software science measure, complexity measures for inter-module and intra-module complexity are obtained. Inter-module complexity should increase significantly with the use of a top-down integration strategy or a bottom-up integration strategy and increase with a leveling off when the sandwich integration strategy is used. Using the G3 big bang integration strategy, inter-module complexity should remain relatively stable. Intra-module complexity should remain stable with little or no change for all integration strategies. One or the other of the two complexity measures should

exhibit a pattern of stability. If neither exhibit stability patterns then the progress of integration is not visibly occurring.

Since none of the patterns in the proposed model require extensive technical analysis in order to show progress during integration, the model can be used by non-technical project personnel as well as members of management and technical personnel. Patterns are easily recognizable by most people. Recognizing the patterns outlined for this model will enhance the visibility of progress during a project's integration phase and enable non-technical and technical project personnel to make rational, data-based decisions about, and projections for, software projects. Appendix C presents a check-list for the model to aid in its use by managers.

Future work to identify models of patterns for the other phases of the software development cycle is needed to build an overall model of patterns of progress for software development. Another area for future research is the examination of the changing of assignment statements to see if the exhibited trend of decreasing percentages holds. A search for a theoretical/logical explanation for this high percentage of change for assignment statements would also be valuable. It is necessary to do further change measure analysis of the integration phase in order to lend credence to the proposed model or to refute it. Evaluation of additional change measures may also expand the proposed model.

As more detailed historical data is gathered about the different software development phases, it is important to remember that there is a need to move to a system for project decision making that is understandable and usable by ALL personnel involved in the software development of the project.

BIBLIOGRAPHY

[Bailey81] John W. Bailey, and Victor R. Basili, "A Meta-Model for Software Development Resource Expenditures", Int. Conf. on Software Engineering (5th, 19810, pp 107 - 116.

[Basili79] Victor R. Basili and Marvin V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment", Computers and Structures, Vol. 10, 1979, pp 39 - 43.

[Boehm81] Barry Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Brooks75] Frederick P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley, Reading, MA, 1975.

[Cheung80] Roger C. Cheung, "A User-Oriented Software Reliability Model", IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, March 1980, pp 118 - 125.

[Doerflinger85] C.W. Doerflinger and V. R. Basili, "Monitoring Software Development Through Dynamic Variables", IEEE TOSE, Vol. 11, No. 9, September 1985, pp 978 - 985.

[Esterling80] R. Esterling, "Software Manpower Costs: A Model", Datamation, March 1980, pp. 164 - 170.

[Fairley85] Richard E. Fairley, Software Engineering Concepts, McGraw-Hill Book Company, New York, NY, 1985.

[Gehring] Philip F. Gehring and Udo W. Pooch, "Toward a Management Philosophy for Software Development", from Advances in Computer Programming Management , Rullo (ed) Hayden.

[Gilbert83] Philip Gilbert, Software Design and Development, Science Research Associates, Inc., Chicago, IL, 1983.

[Gustafson85] David A. Gustafson, Austin Melton, Chyuan Samuel Hsieh, "An Analysis of Software Changes During Maintenance and Enhancement", Conference on Software Maintenance, Washington, D.C., November, 1985.

[Halstead77] Maurice Halstead, Elements of Software Science, Elsevier, 1977.

[Howden82] William E. Howden, "Contemporary Software Development Environments", Communications of the ACM, Vol. 25, No. 5, May 1982, pp 318 - 329.

[Itakura82] Minoru Itakura and Akio Takayanagi, "A Model for Estimating Program Size and its Evaluation", 6th Int. Conf. on Software Engineering, 1982, pp 104 - 109.

[Littlewood73] Bev Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software", Journal of the Royal Statistical Society (series C), Applied Statistics, Vol. 22, No. 3, 1973, pp. 332 - 346.

[Mata84] Ramon Toledo Mata, "A Factor Analysis of Software Complexity Metrics", PhD Thesis, Kansas State University, Manhattan, Kansas 1984.

[Mata86] Ramon Toledo Mata and David A. Gustafson, "A Factor Analysis of Software Complexity Measures", submitted for publication.

[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308 - 320.

[Musa80] John D. Musa, "Software Reliability Measurement", The Journal of Systems and Software, vol. 1, no. 3, 1980, pp. 223 - 241.

[Norden77] Peter V. Norden, "Project Life Cycle Modelling: Background and Application of the Life Cycle Curves", First Software Life Cycle Management Workshop, August 1977, pp. 217 - 306.

[Page82] J. Page, "Evaluation of Management Measures of Software Development Vol. 1 : Analysis Summary", NASA, September 1982, N83 - 13836.

[Pressman82] Roger S. Pressman, Software Engineering - A Practitioner's Approach, McGraw-Hill Book Company, New York, NY, 1982.

[Putnam77] Lawrence H. and Ray W. Wolverton, Quantative Management: Software Cost Estimating, IEEE cat EHO 129-7, New York, 1977.

[Putnam78] Lawrence Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Project", IEEE Transactions on Software Engineering, vol. 4, no. 4, 1978, pp. 345 - 361.

[Putnam79] Lawrence Putnam and Ann Fitzsommons, "Estimating Software Costs", Datamation, Sept - Nov. 1979.

[Putnam80] Lawrence H. Putnam, "The Real Metrics of Software Development", EASCON 80 Record, IEEE cat 80CH 1578-4, New York, 1980, pp 310 - 322.

[Shooman79] Martin L. Shooman, "Tutorial on Software Cost Models", Workshop on Quantative Software Models, IEEE cat TH0067-9, pp 1 - 19.

[Szulewski81] P. A. Szulewski, M. H. Whitworth, P. Buchan and J. B. DeWolf, "The Measurement of Software Science Parameters in Software Design", Perf. Eval. Review, Vol. 10, No. 1, Spring 1981, pp 89 - 94.

[Walston77] C. Walston and C. Felix, "A Method of Programming Measurement and Estimation", IBM Systems Journal, vol. 16, no. 1, 1977, pp. 54 - 73.

[Warburton83] R. D. H. Warburton, "Managing and Predicting the Costs of Real-Time Software", IEEE TOSE, Vol. 9, No. 5, September 1983, pp 562 - 568.

[Weinberg82] G. M. Weinberg, "Overstructured Management of Software Engineering", 6th Int. Conf. on Software Engineering, pp 2 - 9.

[Weiss85] D. M. Weiss and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory", IEEE TOSE, Vol. SE-11, No. 2, February 1985, pp 157 - 168.

APPENDIX A
DATA FOR TEAM G3

TABLE 1
STATEMENT CHANGES BY CHANGE PERIOD - Team G3

STATEMENT TYPE	PERIOD 1	PERIOD 2	PERIOD 3	OVERALL
Subroutine Calls	14.7%	32.9%	16.5%	21.2%
Begin/End	11.1%	21.1%	5.0%	12.3%
Assignment	33.3%	14.3%	10.9%	18.9%
If	13.0%	20.0%	4.3%	12.7%
Else	20.0%	13.3%	6.7%	11.1%
Do	0.0%	20.0%	0.0%	7.1%
Return	0.0%	0.0%	0.0%	0.0%
While	18.2%	9.1%	9.1%	12.1%
Declarations	15.2%	2.3%	4.1%	7.2%
Include	0.0%	0.0%	0.0%	0.0%
Comments	0.0%	33.3%	0.0%	6.5%
Break	0.0%	0.0%	0.0%	0.0%
Case	0.0%	0.0%	0.0%	0.0%

TABLE 2
Raw Data for Types of Statements Changed - Team G3

	# changes	total loc
Assignment	24	127
Begin/End	14	114
Subroutine Calls	48	226
If	9	71
Else	5	45
Do	1	13
Return	0	6
While	4	33
Declarations	10	138
Include	0	6
Comments	3	46
Break	0	0
Case	0	0

TABLE 3
Percent Changes By Module - Team G3

	Delta 1	Delta 2	Delta 3
Main	18.6%	20.0%	11.1%
Findword	89.5%	21.4%	0.0%
Beginproc	11.7%	20.0%	16.2%
Thenproc	15.8%	62.2%	41.9%
Getword	52.6%	0.0%	0.0%
Check	4.1%	9.2%	2.5%
Skip	30.8%	0.0%	0.0%
Declare	0.0%	0.0%	0.0%
Initial	5.3%	0.0%	0.0%

TABLE 4
Complexity Measures - Team G3

McCabe's Procedure Name	Period 1	Period 2	Period 3	Period 4
Beginproc	8	8	8	8
Check	11	12	12	12
Declare	1	1	1	1
Findword	3	3	3	3
Getword	2	3	3	3
Initial	2	2	2	2
Main	5	5	5	5
Skip	2	3	3	3
Thenproc	5	6	3	9
OVERALL	39	44	40	46

Halstead's Procedure Name	Period 1	Period 2	Period 3	Period 4
Beginproc	137	159	178	160
Check	179	193	192	195
Declare	27	27	27	27
Findword	41	49	39	39
Getword	39	53	53	53
Initial	35	35	35	35
Main	36	36	55	61
Skip	11	23	23	23
Thenproc	72	93	59	117
OVERALL	577	669	661	710

APPENDIX B
DATA FOR TEAM G5

TABLE 1
STATEMENT CHANGES BY CHANGE PERIOD - Team G5

STATEMENT TYPE	PERIOD 1	PERIOD 2	PERIOD 3	OVERALL
Assignment	73.3%	9.1%	32.4%	36.0%
Begin/End	0.0%	0.0%	18.1%	6.1%
Subroutine Calls	29.8%	10.6%	76.6%	39.0%
If	22.2%	11.1%	33.3%	22.2%
Else	0.0%	0.0%	40.0%	13.3%
Do	0.0%	0.0%	0.0%	0.0%
Return	0.0%	0.0%	0.0%	0.0%
While	80.0%	0.0%	0.0%	26.7%
Declarations	33.3%	21.9%	45.5%	33.7%
Include	0.0%	0.0%	0.0%	0.0%
Comment	0.0%	1.2%	8.2%	3.2%
Break	100.0%	0.0%	0.0%	33.3%
Case	100.0%	0.0%	0.0%	33.3%

TABLE 2
Raw Data for Totals of Types of Statements Changed - Team G5

	# changes	total loc
Assignment	66	194
Begin/End	12	250
Subroutine Calls	118	292
If	12	54
Else	4	30
Do	0	18
Return	0	0
While	8	30
Declarations	52	170
Include/Define	6	20
Comments	14	504
Break	10	30
Case	10	30

TABLE 3
Percent Changes By Module - Team G5

	Delta 1	Delta 2	Delta 3
Instcode	27.3%	0.0%	18.2%
Firstpass	55.9%	0.0%	21.4%
Secondpass	13.0%	0.0%	13.0%
Mes_proc	12.5%	0.0%	25.0%
Umes_proc	28.0%	0.0%	11.5%
Comment	5.0%	0.0%	25.0%
Insert	13.6%	0.0%	27.3%
Putsymbol	25.7%	0.0%	21.4%
Stack	14.8%	25.9%	32.1%
Push	25.0%	31.8%	66.7%
Pop	15.0%	9.1%	63.6%

TABLE 4
Complexity Measures - Team G5

McCabe's

Procedure Name	Period 1	Period 2	Period 3	Period 4
Comment	2	2	2	2
First_Pass	6	6	6	6
Insert	2	2	2	2
Instcode	1	1	1	1
Mes_Proc	1	1	1	1
Pop	2	2	2	1
Push	2	2	2	1
Putsymbol	1	1	1	1
Second_Pass	3	3	3	3
Stack	4	4	4	4
Umes_Proc	5	5	5	5
OVERALL	29	29	29	27

Halstead's

Procedure Name	Period 1	Period 2	Period 3	Period 4
Comment	29	24	24	30
First_Pass	57	52	52	67
Insert	39	40	40	52
Instcode	13	13	13	15
Mes_Proc	12	13	13	19
Pop	28	35	35	9
Push	28	35	39	9
Putsymbol	15	14	14	17
Second_Pass	41	37	37	40
Stack	37	40	43	35
Umes_Proc	116	124	124	133
OVERALL	415	427	434	426

APPENDIX C

MANAGER'S CHECKLIST FOR PROGRESS EVALUATION

STEP ONE — STATEMENT TYPES BEING CHANGED MOST FREQUENTLY

Choose the group which contains the statement types being changed most frequently.

Declarations, If, While, Else, Begin/End, Case, Break

> Indicates unsuccessful pattern for step one.
> If Declarations are the highest, STOP;
> else proceed to step 2
> PATTERN for step one is "a"

Subroutine/Sytem Calls, Assignment

> Indicates successful pattern for step one.
> Proceed to step 2
> PATTERN for step one is "b"

Comments, Include, Return, Do

> Pattern not established for this model.
> Proceed to step 2
> PATTERN for step one is "c"

STEP TWO — FIND INTEGRATION PATTERN FOR HIERARCHY STRUCTURE

Choose an integration pattern by matching the selection criteria against the observed data.

TOP-DOWN INTEGRATION

Higher percentages of module change at top level modules.
Downward movement of location of higher percentages.
Downward growth of hierarchy structure.
Hierarchy calls stable (if A calls B in early phases
then A calls B throughout).

PATTERN for step two is I
Proceed to step 3 - COMPLEXITY A.

BOTTOM-UP INTEGRATION

Higher percentages of module change at lower level modules.
Upward movement of location of higher percentages.
Upward growth of hierarchy structure.

PATTERN for step two is II
Proceed to step 3 - COMPLEXITY A.

SANDWICH INTEGRATION

Higher percentages of module change at lower levels.
Upward movement of location of higher percentages.
Appearance of total hierarchy.
Higher percentages of module change at top levels.
Downward movement of location of higher percentages.

PATTERN for step two is III
Proceed to step 3 - COMPLEXITY B.

(continued next page)

G3 BIG BANG INTEGRATION

Hierarchy structure is complete and stable.
Higher percentages of module change at lower levels.
Upward movement of location of higher percentages.

PATTERN for step two is IV
Proceed to step 3 - COMPLEXITY C.

NO PATTERN

Higher percentages of module change randomly distributed.
Hierarchy structure may be stable or may be fluctuating rapidly.

PATTERN for step two is V
No visible progress is being made.
Cannot identify an integration technique.
STOP

STEP THREE — COMPLEXITY MEASURE EVALUATION

Complexity A

Inter-module complexity growing significantly.
Intra-module complexity showing slight change but relatively stable.

PATTERN for step three is A

Complexity B

Inter-module complexity growing significantly then leveling off.
Intra-module complexity showing slight change but relatively stable.

PATTERN for step three is B

Complexity C

Both intra-module and inter-module complexity showing slight change but relatively stable.

PATTERN for step three is C

OTHER:

If both McCabe's and Halstead's exhibit a pattern then NO PROGRESS.

If neither McCabe's or Halstead's exhibits a pattern then NO PROGRESS.

PATTERN for step three is D

CONCLUSIONS

Pattern sequences are ranked from those promising the highest probability of success to those with the lowest probability of success.

PROGRESS	STEP 1	STEP 2	STEP 3
Promising	b	I, II	A
	b	III	B
	b	IV	C
	c	I, II	A
	c	III	B
	c	IV	C
	b	I, II	B, C
	b	III	A, C
	b	IV	A, B
	c	I, II	B, C
	c	III	A, C
	c	IV	A, B
Unpromising	a	V	D

A Model of Successful Patterns

of Progress During the Integration of Software

by

MARY LOU A. LANCHBURY

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1986

ABSTRACT

Experience shows that the testing and integration phase of the software development cycle generates the single largest part of the development cost - both in time and money. This single phase accounts for 40 - 50% of the overall development costs and 40 - 50% of the time schedule [Gilbert1983]. Project managers often watch in frustration as projected costs increase and delivery times are delayed. There is a need for a model to evaluate the progress of the testing and integration phase.

This thesis presents a model for evaluating progress during the integration portion of the testing and integration phase. The model patterns concepts that can be understood and used by technical and non-technical managers alike. While it does not purport to be the final ideal in all cases, it is an initial step toward developing a model of progress for the software development cycle.